Ghost in the PLC
Stealth Low-Level Manipulation of PLCs' I/O

Ali Abbasi
University of Twente
Enschede, The Netherlands
Email: a.abbasi@utwente.nl

Majid Hashemi
Quarkslab
Paris, France
Email: m.hashemi@quarkslab.com

Emmanuele Zambon and Sandro Etalle
University of Twente
Enschede, The Netherlands
Email: e.zambon, s.etalle@utwente.nl

***Abstract*—Programmable Logic Controllers (PLCs) are a family of embedded devices used for physical process control. Similar to other embedded devices, PLCs are vulnerable to cyber attacks. Because they are used to control the physical processes of critical infrastructures, compromised PLCs constitute a significant security and safety risk. In this paper, we investigate attacks against PLCs from two different perspectives. We show how to circumvent current host-based detection mechanisms applicable to PLCs by avoiding typical function hooking and by leveraging dynamic memory. We then introduce a specific type of attack against a PLC that allows the adversary to stealthily manipulate the physical process it controls by tampering with the device I/O at a low level. Our study is meant to be used as a basis for the design of more robust detection techniques specifically tailored for PLCs.**

I. Introduction

The security of embedded devices is gaining an increasingly important place in the security panorama. Programmable Logic Controllers (PLCs) are a family of embedded devices that are used in industrial control networks. The security of PLCs is more critical than that of other embedded devices because they control and monitor industrial processes in critical infrastructures [20]. The role of these devices in controlling industrial processes means that the successful exploitation of a PLC can affect the physical world and, as a result, can have dangerous consequences [21]. Despite their criticality, PLCs appear to be equally as vulnerable to cyber attacks as most embedded systems [4], [5], [30]. Today, PLCs include some basic protection mechanisms such as not making the root account available to login users. However, based on recent research [4], [5], [30], one can argue that such protection mechanisms are ineffective. Obtaining system-level access and running arbitrary code on a PLC paves the way for attackers to install malicious software in the PLC. To detect attacks in which the attacker has gained system-level access to embedded devices, several studies have focused on host-based detection techniques [10], [34]. Some of these modern host-based detection techniques that have been introduced for embedded devices can also work for PLCs. We investigated these host-based detection techniques and determined that those that are most practically applicable to PLCs in fact suffer from various design flaws. Most rely on verifying the integrity of the static executable in memory using signatures. Some enforce control-flow integrity (CFI) by taking a black-listing approach rather than ensuring real CFI for performance reasons. Such approaches make us believe that it might be possible to evade all defensive techniques that are applicable to PLCs by designing an attack that exploits the design limitations of these host-based detection techniques for embedded devices.

To the best of our knowledge, no previous research has extensively discussed the design of malicious software (i.e., rootkits) for PLCs. Most research to date has focused on obtaining system-level access to a PLC. Although obtaining system-level access is the essential barrier to the installation of malicious software in a PLC, it is also critical for attackers to understand the underlying system design of the PLC to install their malicious software.

In this paper we evaluate the weaknesses of host-based intrusion detection solutions applicable to PLCs and present a methodology for evading those solutions. We then introduce a new type of attack against PLCs that leverages those weaknesses and allows an adversary to stealthily manipulate the physical process controlled by such a device. The novelty of this attack lies in the fact that we target a layer which has never been tackled before in PLCs. The attack, instead of targeting the PLC logic [16], [24], [25] or firmware [4], [5], [9], [39], manipulates the way the firmware interacts with the I/O.

The remainder of this paper is organized as follows: In Section II, we describe the state of the art in attacks against and defenses for embedded devices. In Section III we discuss the parameters of an applicable host-based defensive solution for PLCs. In Section IV, we describe a methodology for bypassing two defensive solutions for embedded devices. In Section V, we introduce our I/O attack against PLCs. We describe the practical implementation of the proposed I/O attack in the form of a rootkit in Section VI. In Section VII, we discuss the limitations of the I/O attack and possible detection mechanisms for the attack methodology developed in Section IV. Finally, we draw our conclusions and lay a foundation for future work in Section VIII.

II. Background

A. Attack Techniques

The attack techniques used against embedded devices can be divided into three categories: (i) firmware modification attacks, (ii) configuration manipulation attacks and (iii) control-flow attacks.

- Firmware modification attacks: in recent years, a number of firmware modification attacks against embedded devices have been researched and discussed. Cui et al. [9] demonstrated how the HP-RFU firmware update protocol can be exploited to allow adversaries to inject malicious firmware into HP printers. Traynor et al. [37] showed how to recursively compromise embedded devices and use them to create a network of malicious devices by manipulating their firmware. Wegner [39] demonstrated how to install a backdoor into Siemens office telephone communication devices by exploiting a vulnerability in their firmware verification system. Basnight et al. [4] illustrated that it is feasible to execute arbitrary code in a PLC by exploiting the firmware update feature, and finally, Peck et al. [29] showed how to exploit the Ethernet module of a PLC by uploading malicious firmware to it.
- Configuration manipulation attacks: these attacks allow an adversary to modify critical configuration parameters of an embedded device to force it to misbehave. For example, an anonymous security researcher with the nickname PT [31] demonstrated how to obtain access to a Private Branch Exchange (PBX), an embedded device used for telephone systems, by exploiting a vulnerability in the proprietary authentication protocol used by one vendor. A special case of configuration manipulation attacks concerns programmable devices, such as PLCs. PLCs can be programmed to control a physical process by following the logic specified by the user. In this case, the attack consists of uploading a malicious logic to alter the manner in which the process is controlled. Falliere et al. [16] reported that the Stuxnet malware was used to manipulate the logic of PLCs from a programming station to subvert part of the uranium enrichment process at Natanz (Iran). In [24], [25], McLaughlin et al. introduced two techniques for the dynamic generation of a malicious PLC control logic. To the best of our knowledge, the techniques proposed by McLaughlin et al. are, for the moment, limited in their practical applicability and have never been used in real-world attacks.
- Control-flow attacks: in general, this category of attacks consists of manipulating the execution flow of a running process. This is typically achieved by exploiting a stack/heap overflow or use-after-free vulnerability, which allows for the execution of arbitrary code by an adversary. Jump- and return-oriented programming (JOP and ROP) are considered to be control-flow attacks. Recent research has illustrated the possibility of control-flow attacks in embedded devices. For example, Beresford [5] presented multiple protocol vulnerabilities in Siemens PLCs that can allow an adversary to perform a remote code execution attack. Wightman demonstrated that Schneider Electric PLCs are vulnerable to buffer overflow attacks [19], [40]. Heffner [27], [32], [33] presented multiple memory corruption vulnerabilities in home routers.

  Although several techniques have been proposed to detect or prevent control-flow attacks on general IT systems, this class of attacks remains one of the most dangerous. Effective countermeasures that are simultaneously applicable in the domain and not circumventable by adversaries have yet to be developed. For example, Schuster et al. [35] evaluated several detection techniques for control-flow attacks [7], [18], [28] and claimed that attackers can bypass them using the code sequence within the executable modules of the target program. Davi et al. [12] introduced several techniques for bypassing detection techniques for control-flow attacks in multiple system security products [18], [26], [28]. Specifically, they showed not only that adversaries can find sufficient ROP gadgets within a program's binary code but also that by using long loops of NOP gadgets, they can create a long gadget chain and thereby break detection mechanisms for control-flow attacks.

B. Detection Techniques

We distinguish three main categories of techniques that have been proposed in the literature for host-based detection of attacks in embedded systems: (i) firmware integrity verification, (ii) memory verification and (iii) control-flow integrity.

- Firmware integrity verification: verifying the integrity of firmware allows one to detect or prevent firmware modification attacks. Such verification can be performed by the host when storing new firmware or at runtime.

  Adelstein et al. [2] introduced a firmware-signing method that consists of a "certifying compiler" for firmware. The compiler allows the firmware to be verified at runtime by checking certain properties of the execution flow, memory and stack integrity in the firmware. Zhang et al. [41] introduced IOCheck, a framework to verify at runtime the integrity of firmware and the I/O configuration of computer I/O peripherals. After a (assumed trusted) BIOS boot, IOCheck leverages the System Management Mode of x86 CPU architectures to perform integrity checks that can be either executed at random polling intervals or driven by specific events. Finally, Duflot et al. [15] introduced NAVIS, a framework for the detection of firmware integrity manipulation in the memory of a network card by inspecting the memory accesses performed by the NIC processor against a model of expected behavior based on the memory layout profile of the adapter. A memory access that is outside the NIC memory profile is interpreted as an attempt to manipulate the NIC firmware.
- Memory verification: these techniques verify the integrity of executable code in memory at runtime. The most common technique for memory verification is attestation, which is used for low-power embedded devices. Attestation is a challenge-response technique that allows an external application (the verifier) to verify the integrity of (parts of) the state of a system (the prover) against malicious modifications. Attestation techniques typically require the availability of dedicated hardware (e.g., a Trusted Platform Module). However, because of the

practical limitations in embedded devices, certain works have focused on the development of pure software-based attestation techniques.
Seshadri et al. [36] introduced SWATT, a software-based attestation technique that can remotely verify the runtime memory contents of embedded devices and discover malicious modifications. SWATT uses a challenge-response protocol to remotely control the memory content of the embedded devices. LeMay et al. [22] proposed an ad hoc static kernel for smart meters that can cryptographically sign every new firmware version uploaded to a device. The signature is sent to the verifier to attest that the current (and previous) firmwares loaded on the smart meter are legitimate and integer. Armknecht et al. [3] introduced a framework for evaluating the security of software-based remote attestation techniques. The authors discussed the security properties of common basic cryptographic functions, such as pseudo-random number generators (PRNGs) and hash functions, when used for attestation purposes. They also discussed the possibility of leveraging time as a verification parameter to strengthen the security of an attestation scheme.
In an approach different from that of memory attestation frameworks, Cui et al. [10] proposed a new host-based deployment mechanism for embedded devices running operating systems, which they called a Symbiotic Embedded Machine or symbiote. The mechanism is specifically designed to inject intrusion detection functionality into the firmware of such devices and to verify the integrity of its executable parts. A symbiote is a code structure embedded in a piece of firmware that can closely co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The symbiote is embedded in a randomized fashion to protect itself from removal, and the execution context of the symbiote is separated from that of the operating system to make it more resistant against adversaries. The authors demonstrated the deployment of a symbiote in the Cisco IOS firmware, with a low performance penalty and without an impact on the router's functionality. Symbiotes cannot continuously monitor the entire firmware but rather set specific watchpoints and monitor certain executable locations of the firmware.

- Control-flow integrity: the vast majority of control-flow hijacking attacks operate by exploiting memory corruption bugs, such as buffer overflows, to control an indirect control-flow transfer instruction in the vulnerable program, most commonly a function pointer or return address. CFI mechanisms counter control-hijacking attacks by ensuring that the control flow remains within the control-flow graph (CFG) intended by the programmer.
  In the context of CFI approaches for embedded devices, Reeves et al. [34] introduced a host-based intrusion detection system for embedded devices that leverages a built-in kernel tracing framework to identify control-flow anomalies in syscalls. The system is constructed by learning, for each monitored syscall, a list of known good source addresses. During detection, the system checks that when a certain syscall is invoked, the source of the call is on the safe list. Although its detection capabilities are limited, the approach also imposes a limited overhead on the system, which makes it suitable for being deployed in embedded devices such as those used in power grids (RTUs, IEDs and PLCs). Other CFI approaches for embedded devices are hardware-assisted. Special hardware modifications to the devices are needed to support the proposed approaches. The authors motivate the need for hardware modifications by citing the limited processing capabilities of embedded devices or the lack of features required by existing CFI approaches (e.g., memory management units or execution rings) in simpler, low-cost, embedded devices. Abad et al. [1] introduced a hardware-assisted CFI system for embedded devices. The system employs a dedicated hardware component to compare the control flow of the embedded device firmware at runtime to the CFG. The graph is constructed by decompiling the binary of the application to be protected. However, the method proposed for constructing the CFG does not consider indirect control-flow transfers (e.g., indirect function calls); therefore, the approach is incomplete and prone to certain types of control-flow attacks in which the adversary manipulates the control flow of the target application by changing the values of data memory areas. For example, the adversary may first spray the heap memory with shellcode instructions and then overwrite the value of a function pointer to point to a random heap address, which may contain the shellcode. Francillon et al. [17] proposed a hardware-assisted protection mechanism for AVR microcontrollers against control-flow attacks. The mechanism consists of separating the stack into a data stack and a control stack. The control stack is hardware-protected against unintended or malicious modifications (i.e., those not performed by call or ret instructions). Finally, Davis et al. [11] proposed a hardware-assisted CFI scheme that uses the hardware to confine indirect calls. This CFI scheme is based on a state model and a per-function CFI labeling approach. In particular, the CFI policies ensure that function returns can only transfer control to active call sides (i.e., return landing pads of currently executing functions). Furthermore, indirect calls are restricted to target the beginning of a function, and finally, behavioral heuristics are used to address indirect jumps.

III. Detection Mechanisms Applicable to PLCs

Not all of the defensive techniques described in Section II-B are practically applicable to embedded control devices such as PLCs. We consider two primary parameters to determine which defensive solutions are, in fact, practical. These parameters are as follows:

- Designed for embedded devices that run modern operating systems: there is a group of embedded devices, called low-powered embedded devices, that do not have an operating system (OS). Devices that run microcontroller-based processors (such as AVR or ATMEL) can be considered as low-powered embedded devices. However, most of the PLCs have a *real* OS. Therefore we only consider approaches that target embedded devices with a *real* OS.
- No hardware modification: the performance limitations make it difficult to introduce a complete host-based security mechanism for PLCs. Most of the solutions described in Section II attempt to overcome these limitations by first considering hardware modifications of the embedded devices, thus making those solutions less attractive.

Based on the parameters, we can identify two host-based detection mechanisms that, unlike most of the techniques described in Section II-B, possess both desired qualifications. These host-based detection systems are Autoscopy Jr. [34] and Doppelganger [10]. For the sake of completeness, we use a third condition, namely CPU overhead, for applicable host-based detection systems for PLCs. In embedded devices, high CPU overhead is an important issue for host-based detection systems. Large CPU overhead makes host-base defenses for embedded systems less practical since the CPU resources in such systems are very limited. We consider the same threshold achieved by the authors of Autoscopy Jr. [34] as an acceptable overall CPU overhead. Considering the CPU overhead limit as a new condition, leads to the same selected defensive solutions since both satisfy this new requirement.

These solutions are practically applicable, and because of their practical approach, they were adopted in the industry immediately upon their introduction [8], [34]. However, these approaches also exhibit certain weaknesses. Understanding these weaknesses assist us in designing better host-based solutions for embedded devices.

- Autoscopy Jr.: Autoscopy Jr. is a kernel control-flow monitoring system that searches for control-flow anomalies caused by function hooking in the kernel [34]. Autoscopy Jr. incurs only 5% CPU overhead, which is a significant achievement for a host-based detection system for embedded devices. Autoscopy Jr. specifically searches for kernel attacks in which the malicious code manipulates a function pointer. When a process calls a function with a manipulated pointer, the call is diverted to the malicious function instead of a legitimate one. The malicious function can then decide either to never call the original function or to call the legitimate function with a manipulated input. Autoscopy Jr. operates in two phases:
  1) Learning phase: In the learning phase, Autoscopy Jr. installs a character device driver that allows it to access the kernel memory by invoking `ioctl()`. Next, Autoscopy Jr. uses the device driver to monitor direct and indirect function calls and their corresponding return addresses. Afterward, it saves the return addresses of these functions, with certain runtime information (such as function arguments), to a data structure called the Trusted Location List (TLL). It then uses the TLL during the detection phase.
  2) Detection phase: During the detection phase, Autoscopy Jr. uses the previously installed device driver to monitor function calls. When a function that is listed in TLL is called, Autoscopy Jr. verifies the function address against the TLL entry for the same function. If the function address is not found in the TLL, it generates an alert.

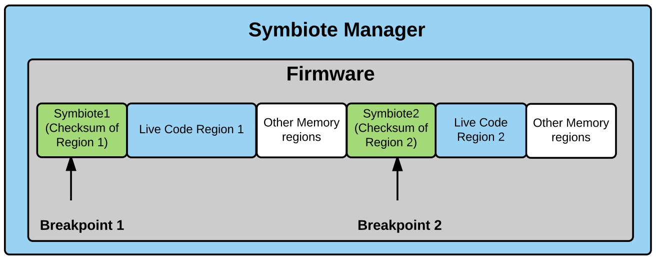

Fig. 1. Structure of embedded device firmware controlled by Doppelganger

- Doppelganger: Doppelganger is a host-based intrusion detection solution for embedded devices. It can detect both kernel- and application-level attacks in embedded devices. Doppelganger first analyzes the firmware of the embedded device to detect live code regions therein. Live code regions are executable parts of the firmware. Once Doppelganger detects the executable area of the memory, it randomly inserts its symbiotes (watchpoints) into the detected live code areas. Doppelganger symbiotes contain a CRC32 checksum of the randomly selected live code regions.
  Doppelganger adds its symbiote manager to the beginning of the firmware. The symbiote manager can be regarded as a debugger that runs the firmware of the embedded system. The symbiote manager causes Doppelganger to run in a different context of the OS to make it resistant to attacks against its runtime. During the firmware execution, every time the symbiote manager detects a symbiote in memory, it stops the execution process (treating it as a breakpoint) and compares the current CRC32 checksum of the memory area with the symbiote checksum. If the checksum does not match, Doppelganger considers this finding to be evidence of a code modification attack and does not allow the processor to continue running the code. Figure 1 depicts the structure of embedded device firmware consisting of a symbiote manager and symbiotes.

IV. Methodology for Evading Defensive Mechanisms

Both Autoscopy Jr. and Doppelganger provide practical host-based intrusion detection mechanisms for embedded de-

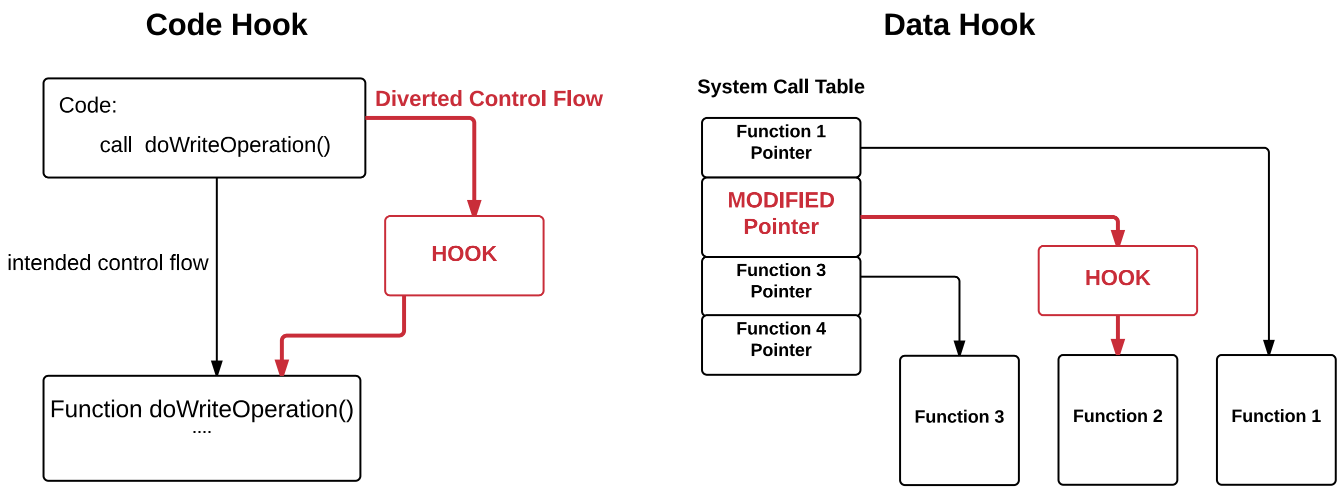

Fig. 2. Typical function hooking

vices with little performance overhead that can be applied to PLCs. Autoscopy Jr. detects kernel control-flow violations, and Doppelganger detects code modifications at runtime. However, both the Autoscopy Jr. and Doppelganger approaches suffer from certain shortcomings. These shortcomings can be divided into three types, each of which applies to at least one of the two approaches.

- Static referencing: Both Autoscopy Jr. and Doppelganger use static references to verify the execution flow or the integrity of an executable code region. Static referencing is comparable to signature-based approaches. If an attacker avoids the explicitly defined references, he can evade detection.
  The static references in Autoscopy Jr. are the entries of the TLL. In Doppelganger, the static references are the symbiotes. None of these references can be modified during runtime. Autoscopy Jr. requires an additional learning phase to add more entries to the TLL, and Doppelganger requires the recreation of the firmware to insert additional symbiotes. These requirements limit the capabilities of both Doppelganger and Autoscopy Jr.: if an attacker inserts malicious code into locations that are not considered among the static references, then this malicious code can not be detected.
- Function hooking: In general, there are two types of function hooks: *code hooks* and *data hooks* [23], [38]. Both types of hooks are illustrated in Figure 2. In code hooking, an attacker can divert function calls by modifying executable parts of the kernel, such as the *.text* section. If the attacker wishes to hook the function call `doWriteOperation()`, as illustrated in Figure 2, he modifies the executable instructions that call `doWriteOperation()` to instead call its hook.
  In data hooking, the attacker does not manipulate executable instructions; instead, he modifies the function pointers in the System Call Table (or other similar tables, such as the System Service Dispatch Table) to call their hooks. The System Call Table consists of pointers to system call functions. If an attacker modifies a function pointer and that function is then called by a process, the OS calls the *hook function* instead of the original function.
  Unfortunately, Autoscopy Jr. detects only data hooks and is unable to prevent code hooking attacks. Moreover, because Autoscopy Jr.'s approach to detecting data hooking is not complete, an attacker can define his own versions of functions and call them separately. Autoscopy Jr. does not generate alerts for such unknown function calls since they are not functions that are listed in the TLL.
- Dynamic memory: Doppelganger sets its watchpoints prior to execution in the static executable parts of the firmware to be protected. We can compare the Doppelganger detection mechanism to code hooking detection mechanisms. Code hooking detection mechanisms search for modifications in the static parts of a kernel or application. This is a very similar approach to that of Doppelganger. Since it monitors only the static executable parts of the memory, Doppelganger is vulnerable to dynamic memory modification attacks (e.g., heap overflows). Doppelganger cannot detect any attack originating from dynamic memory.
  The authors of Doppelganger claim that in their future research, it might be possible to verify the integrity of dynamic memory. Although verifying the integrity of dynamic memory might be possible, we argue that the detection mechanism they propose cannot be extended to dynamic memory since it is based on static information (the CRC checksum of the memory area). Therefore, it is not a straightforward extension to also monitor the content of dynamic memory.

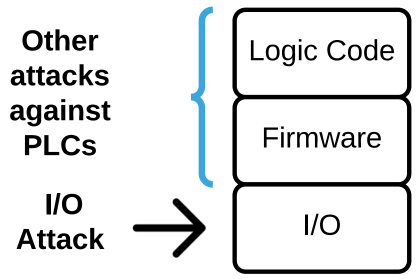

Fig. 3. Comparison between I/O attack and other attacks against PLCs

Using a Loadable Kernel Module (LKM) is one of the methods an attacker can use to gain access to the kernel space to install a rootkit. The kernel uses `vmalloc()` to allocate LKMs into the heap area of the memory, which is dynamic memory. This type of allocation makes the executable instructions of a rootkit completely invisible to Doppelganger because it is not searching in dynamic memory.

Doppelganger, in its current implementation, can be bypassed when an attacker inserts malicious code into a part of the memory that contains dynamic contents. We call these parts of the memory *dynamic content memory*.

Dynamic content memory regions are memory regions that are statically allocated but whose contents can change dynamically. As a result, Doppelganger cannot create a checksum of these memory regions. An example of dynamic content memory is Thread-Local Storage (TLS). At the beginning of the execution of a process, the OS allocates a fixed chunk of memory for the TLS, but the TLS contents is used as dynamic content memory for temporary variables and data during the process. If an attacker inserts malicious code into the TLS and executes it from the TLS, Doppelganger will not be able to detect this malicious code execution because of the dynamic nature of the TLS.

One might assume that a combination of Autoscopy Jr. and Doppelganger could provide sufficient protection to detect both data hooking and code hooking. However, we have found that it is still possible to craft an attack that will go unnoticed even when both approaches are used in combination.

V. A New Kind of Attack

In Section II-A, we introduced known attacks that can be used against PLCs. In this section, we describe a new type of attack that targets PLCs based on the methodology described in Section IV. PLCs are embedded devices that are sensitive components of critical infrastructures and are used in various industrial environments to control physical processes. Because of the manner in which PLCs operate, we have identified new possible means for attackers to exploit them.

We assume one of the main goals to attack a PLC is to manipulate the physical process by sending signals to the sensors and actuators controlled by the PLC, while simultaneously remaining undetectable to the PLC logic, firmware, and its operators. Physical process manipulation can have serious consequences for the safety of equipment and human life. For example, an adversary may manipulate the value of tank pressure sensors in a pressure sensitive boiler thus leading to the explosion of the boiler, or, similarly to Stuxnet, change the frequency of variable speed drives of centrifuges in a uranium enrichment facility, leading to damage of the centrifuge cascades.

The novelty of our attack lies in the fact that to manipulate the physical process we do not modify the PLC logic instructions or firmware [4], [5], [9], [24], [25], [39]. Instead, we target the interaction between the firmware and the PLC I/O (see Figure 3). This can be achieved without leveraging traditional function hooking techniques and by placing the entire malicious code in dynamic memory, thus circumventing detection mechanisms such as Autoscopy Jr. and Doppelganger. Additionally, the attack causes the PLC firmware to assume that it is interacting effectively with the I/O while, in reality, the connection between the I/O and the PLC process is being manipulated.

A. PLC operation

The main components of a PLC firmware is a software called runtime. The runtime software interprets or executes another code (or executable) known as the *logic*. The logic is a compiled form of the PLC's programming language, such as function blocks or ladder logic. Ladder Logic and Function Block Diagrams are graphical programming languages that describe the control process. A plant operator programs the logic and can change it when required. The logic is dynamic code, whereas the runtime software is static code. The purpose of a PLC is to control equipment, and to do so, it must interact with its I/O. The first requirement for I/O interaction is to map the physical I/O addresses into memory. The drivers or PLC runtime map the I/O memory ranges. Additionally, at the beginning of logic execution, the PLC runtime software must initialize the processor registers related to the I/O used in the logic. During the initialization, the appropriate modes for the I/O are set by the runtime software.

For example, it sets the "output" mode for I/O pins that are used for write operations in the logic or the "input" mode for I/O pins that are used for read operations in the logic. This stage is called the *I/O initialization sequence*.

After I/O initialization, the PLC runtime software executes the logic in every run cycle. In an ideal scenario, we can assume that the PLC prepares the logic execution by scanning the logic inputs (e.g., the I/O inputs that are used in the logic) and the set points from the variable table. The variable table is a virtual table that contains all set points and input or output variables used within the logic. The operation that consists of reading the inputs, executing the logic code, and updating the outputs is called the program scan. During the program scan, any changes in the variable table (for the I/O inputs) are ignored until the beginning of the next program scan. At

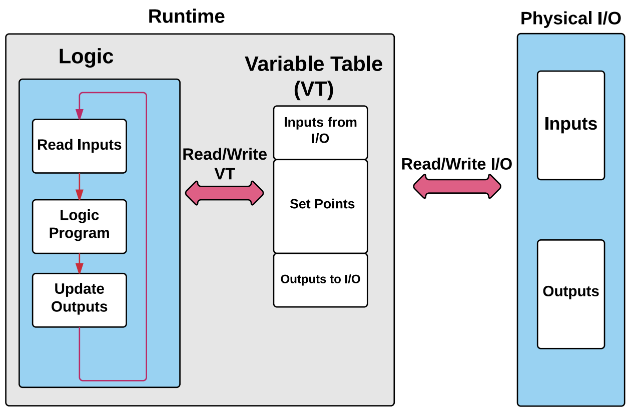

Fig. 4. Overview of PLC runtime operation, the PLC logic and its interaction with the I/O

the end of the program scan, the PLC runtime software writes to the related part of the mapped memory that eventually is written to the physical I/O by the kernel. Figure 4 depicts the PLC runtime operation, the running of the logic, and its interaction with the I/O.

At hardware level, a PLC typically comprises separate digital and analog inputs and outputs. Because PLCs, similar to other computers, are digital systems, they cannot control analog input and output without additional hardware components. Digital-to-Analog Converters (DACs) for analog outputs and Analog-to-Digital Converters (ADCs) for analog inputs form part of the analog interface of a PLC. These components read or write analog values by converting them to or from digital outputs or inputs to allow the PLC to interact with its analog interfaces. The DACs and ADCs are not separate components of the PLC but rather an integral part of the PLC circuit board. One can argue that the basis of I/O interaction in PLCs is digital. Analog control is simply a conversion of digital signals into analog signals or analog signals into digital signals.

B. I/O attack

We assume that to perform the *I/O attack*, the attacker can gain root access to the targeted device. This can be achieved through firmware verification attacks, through control-flow attacks against the PLC runtime, or by guessing default passwords. By installing rogue firmware into the PLC, the attacker can infect every binary in the PLC. This can give the attacker complete leverage over the PLC operating system. Using a control-flow attack, the attacker can gain access at the same level as the PLC process. Previous research has revealed that various PLCs run their runtime software as the root user by default [5], [13].

In the case that the PLC runtime is vulnerable to control-flow attack but is not running as root, the attacker needs a privilege escalation vulnerability to gain root access to the PLC.

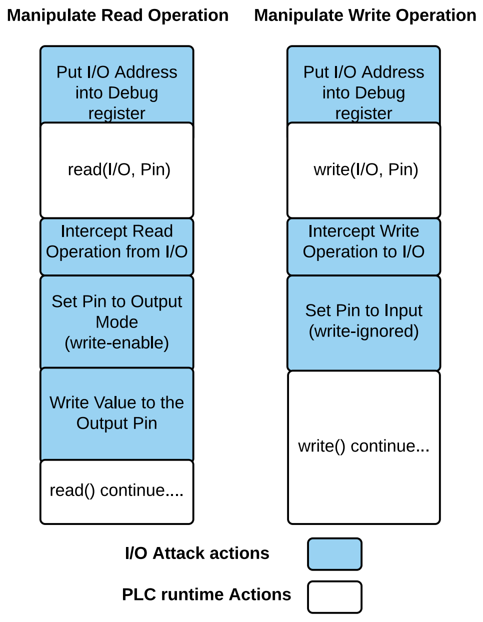
Manipulate Read Operation
Manipulate Write Operation
Put I/O Address into Debug register
read(I/O, Pin)
Intercept Read Operation from I/O
Set Pin to Output Mode (write-enable)
Write Value to the Output Pin
read() continue....
Put I/O Address into Debug register
write(I/O, Pin)
Intercept Write Operation to I/O
Set Pin to Input (write-ignored)
write() continue...
I/O Attack actions
PLC runtime Actions

Fig. 5. Steps of the I/O attack for read and write manipulation

Regarding default passwords, several reported vulnerabilities suggest that some vulnerable PLCs have default root passwords [6], [14]. An attacker can log in to such a device using the default root password to execute his attack.

In addition to the root access requirement, in case the host OS does not provide the physical I/O addresses of the drivers (e.g. via /proc/modules), the attacker needs to know the CPU version of the PLC to investigate it and obtain the physical memory locations for the I/O. We discuss this requirement later in Section VI.

We assume that the attacker already knows the physical process and is aware of the mapping between the I/O pins and the logic. The PLC logic might use various inputs and outputs to control its process; thus, the attacker must know which input or output must be modified to affect the process as desired. The work presented by McLaughlin et al. [24], [25] can be used to discover the mapping between the different I/O variables and the physical world.

The objective of an I/O attack is not merely to manipulate the I/O, but rather to tamper with the I/O in a stealthy manner that does not use typical rootkit techniques. To execute our I/O attack, we must be able to intercept the I/O read or write operations. If we were to use conventional function hooking techniques, most Control-Flow Integrity (CFI) mechanisms would be able to detect our attempts. Therefore, we designed our attack such that it does not use typical function hooking techniques, but can intercept read or write operations. We use the processor debug registers for our attack. Debug registers were introduced to assist developers in analyzing their software, and all new processors with various different architectures (ARM, Intel, and MIPS) have such registers. These registers allow us to set hardware breakpoints to specific memory addresses. Once an address that is in the debug register is accessed by a process, the processor interrupt handler is called and runs a customized interrupt handler.

As the first stage of our attack, we set the mapped I/O addresses to the debug register and intercept every write or read operation using them. In the second stage of our attack, we exploit the I/O initialization sequence. Every I/O, before being used by an application, must be initialized. One step of this initialization process is to define the input/output state of the I/O. If a program wants to use a specific pin in the I/O, it must declare its input/output state to the processor. We manipulate this initialization sequence in our *I/O attack* to change the state of the I/O at runtime whenever the system performs a read/write operation. By intercepting read/write operations using debug registers and manipulating the I/O initialization sequence, an attacker can manipulate the I/O without using any conventional function hooking technique (i.e., code or data hooking) and without being detected by current host-based solutions for embedded devices. Additionally, the attacker does not need to manipulate the PLC system status reports that are being sent to the SCADA software because even the PLC runtime software itself is not be aware of the I/O manipulation.

The attacker first intercepts the write and read operations of the I/O by inserting his desired addresses into the debug registers of the processor. As described earlier, this allows the attacker to intercept the read/write operations and to call his own interrupt handler after interception. When the PLC runtime software wants to read from or write to the I/O, the processor halts the process and calls the attacker interrupt handler. Depending on the type of runtime operation (i.e., read or write) being performed in the I/O, the attacker can decide how to proceed as follows:

1) For write operations: If the PLC runtime software is attempting to write a value to an I/O pin that is initialized as output, the attacker can reinitialize the I/O pin in an input state and allows the runtime software to continue its operation. The runtime software then attempts to execute its write operation to the I/O pin, which has been reinitialized as input. However, the processor ignores such write operations to the I/O because the I/O has been reinitialized in input mode. Figure 5 depicts the manipulation of write and read operations in an I/O attack.
2) For read operations: When the PLC runtime software is attempting to read a value from an I/O pin that is initialized as input, the attacker can reinitialize the I/O pin as an output pin, allowing him to write the value that he wishes to feed to the PLC runtime software into the reinitialized I/O pin. The attacker can then either switch the state of the pin to input mode or allow the runtime software to read the value from an output-mode pin.

Under an I/O attack, the PLC runtime reads the values desired by the attacker from the I/O. The runtime software is not able to write to the I/O; instead, the attacker writes his desired values.

VI. Experimental Demonstration of the I/O Attack

To demonstrate the feasibility of the I/O attack, we implemented a rootkit and attempted to manipulate a representative physical process.

A. Target Device and Runtime

We chose a Raspberry Pi 1 model B as our hardware for the experiment because of the similarity in CPU architecture, available memory, and CPU power to a real PLC. The Raspberry Pi 1 uses a Broadcom BCM2835 single-core processor with a clock speed of 700 MHz. We also used the Codesys platform as a runtime environment. Codesys is a PLC runtime environment that can execute Ladder Logic or Function Block languages on proprietary hardware and provides support for industrial protocols such as Modbus and DNP3.

Currently, more than 260 PLC vendors use Codesys as the runtime software for their PLCs [13]. The combination of features offered by the Raspberry Pi and the Codesys runtime environment make such a system an alternative to low-end PLCs.

The Raspberry Pi includes 32 general-purpose I/O pins, which represent the PLC's digital I/O pins. These digital I/Os can also control analog devices by means of various electrical communication buses (e.g., SPI, I2C, and Serial) available for the Raspberry Pi with external hardware such as ADC or DAC circuit boards.

B. The Logic and the Physical Process

We use pins 22 and 24 of our Raspberry Pi to control our physical process. In our logic, we declare pin 22 as the output pin and pin 24 as the input pin. When the Codesys runtime environment receives our logic, it initializes I/O pin 22 as an output pin and I/O pin 24 as an input pin in the processor registers related to I/O. In the physical layout, our output pin is connected to an LED and our input pin is connected to a button.

The electrical current becomes disconnected when the button is pressed and is reconnected when the button is released. From the perspective of Codesys, if no one is pressing the button, the input pin returns a value of TRUE to the runtime environment, whereas if someone is pressing the button, Codesys receives a value of FALSE from the pin. The pseudocode for the logic that controls these two I/Os is illustrated in Algorithm 1. According to our logic, the LED turns on or off every five seconds. If someone is pressing the button, the LED simply maintains its most recent state until the button is released, at which time the LED again begins to turn on and off every five seconds.

C. Interaction Between the Virtual I/O Registers and the Runtime Environment

In this section, we briefly describe how the virtual I/O registers operate in a BCM2835 processor. Additionally, we describe how the Codesys runtime environment interacts with these registers. The virtual I/O registers are simply I/O address

```
input  : State of $In.24$
output: State of $Out.22$
Main Logic;
while True do
    read input;
    while input True do
        switch_state(output, five seconds);
        //states are High or Low.
    end
    if input False then
        hold the state of the output;
    else
        go to first while;
    end
end
```

Algorithm 1: Logic for our representative physical process

ranges in the processor memory that perform different types of I/O operations. The operation types, the memory sizes, and the physical addresses of the three virtual I/O registers related to read, write, and I/O initialization operations in the BCM2835 processor are summarized in Table I.

TABLE I
I/O MEMORY MAP IN THE BCM2835 PROCESSOR

Operation	Size of each pin in the register	Address of Pin 0	Address of Pins 22 and 24
Input/Output Mode register	3 bits	0x20200000	0x20200002
SET register	1 bit	0x2020001C	0x2020001C
READ register	1 bit	0x20200034	0x20200034

The three virtual I/O registers that are used in our logic are as follows:

1) Input/Output Mode register: This register sets pins to input or output mode. Each pin in this register has three bits of space. The first bit of each pin's bit space defines whether the pin is used for input or output. If the first bit for a pin is set to 0, the pin is an input pin. If the first bit is set to 1, the pin is an output pin. The Codesys runtime software cannot change the value of a pin in input mode; it can only read from it. Even if Codesys writes a value to an input-mode-enabled pin, the operation has no physical effect and is ignored by the processor. However, the Codesys runtime software can change the value of a pin when the pin is in output mode. The input/output mode address range begins at offset 0x2020000 and ends at address 0x20200002. For pin 22, bits 6 to 8 of address 0x20200002 are used.
2) SET register: If a pin is declared as an output pin in the Input/Output Mode register, then every write operation related to this pin in the SET register address can immediately set the pin to high or low. High means that the Raspberry Pi directs an electrical current to the pin, and low means that the electrical current is disconnected from the pin. Every pin in this register has a 1-bit space. Assuming that pin 22 is in output mode, setting a value of "0" or "1" in bit 21 (bit 0 for pin 1, bit 21 for pin 22) of this register causes pin 22 to be set high or low (turning the LED on or off), respectively. The physical address for this I/O register is 0x2020001C.
3) READ register: The Codesys runtime software can read the values of the pins from the READ register. Every pin in this register has a 1-bit space. The values in the READ register have a direct relation with the values in the SET register. For example, if the Codesys runtime software writes a value of "1" to the SET register associated with pin 22 when this pin is initialized as output mode, then the READ register value for the corresponding pin is updated to "1" as well.

To explain how the Codesys runtime environment interacts with the virtual I/O registers, we describe a write operation for pin 22 executed by the Codesys runtime software. We only describe a write operation because read and write operations follow similar procedures. Figure 6 depicts a write operation for pin 22 executed by the Codesys runtime software. During the startup of the Codesys runtime environment, the OS maps the addresses of the I/Os into the Codesys TLS in the user space. This is a special case because the I/Os of the Raspberry Pi are accessible to user-space applications. In other devices, the I/Os might be mapped into the TLS or global variable table of the drivers in the kernel space; however, this does not change the principle on which our I/O attack operates. Once the I/Os are mapped, the mapped I/O addresses are recorded in the page table and a cache that is called the Translation Lookaside Buffer (TLB). The page table is a structure in which the OS maintains the list of mapped physical addresses and their corresponding virtual addresses for the process. The page table can become so large that searching it can be time consuming for the OS. To avoid this scenario, the OS also uses a cache for the page table, which is the TLB.

After our logic is uploaded to the Raspberry Pi, the Codesys runtime software evaluates the I/O description in the logic to determine which pins are designated as input or output pins. Pins 22 and 24 are used as output- and input-mode I/Os in our logic. Because pin 22 is declared as an output pin in our logic, Codesys performs a write operation in the Input/Output Mode register and set bits 6 to 8 of offset 0x20200002 to 100. To execute this write operation, the Codesys runtime asks for the virtual address of 0x20200002. The OS look up the virtual address of 0x20200002 (mapped address of 0x20200002) in the TLB and page table. In our experiment, this address (as expected) was located in the TLS memory area of the Codesys runtime environment with the value 0xB6FCD01A. Once Codesys has retrieved the virtual address, it writes to it and returns to the Codesys runtime process. Codesys then knows that the I/O initialization sequence was successful and that the I/O pins are ready to use.

The Codesys runtime software then begins to execute the logic. When the logic updates the value for pin 22 to "1"

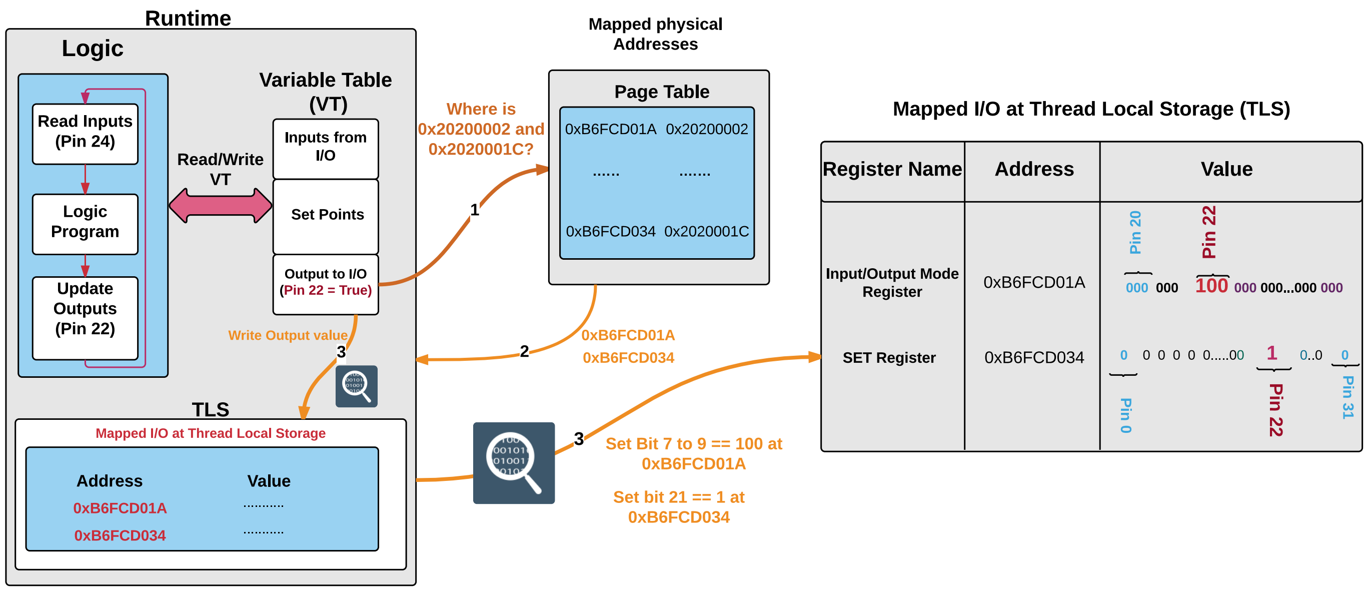

Fig. 6. A search for the address of Pin 22 and a write operation for it

(high) to turn on the LED, the Codesys runtime software must write a value of "1" to bit number 21 of the address 0x2020001C in the SET register. However, Codesys needs to know the virtual address of 0x2020001C. Therefore, the Codesys runtime software looks in the TLB and page table for the mapped address of 0x2020001C. In our case, the address was located in the address 0xB6FCD034. Once the OS has found the virtual address, a value of "1" is written to the register and a *success* result is returned to the related function in the Codesys runtime environment. The Codesys runtime software then updates the I/O state in the SCADA or Human Machine Interface (HMI) software and reports that pin 22 is high (the LED is on).

D. Rootkit Implementation

The implementation is based on the representative logic described in Section VI-B. The objective of our rootkit is to alter the read and write operations of the Codesys runtime environment.

The implementation does not hook the Codesys functions responsible for reporting the I/O status. It does not manipulate the function pointers, nor does it change the original Codesys instructions responsible for updating the I/O (no code hooking). The implementation instead uses the debug registers and I/O initialization sequence to deceive the Codesys runtime into believing that it has successfully written to the I/O or read the current value from the I/O. For example, although the Codesys runtime sets pin 22 to the high (True) state and reports to the SCADA server that the pin state is high (True), the rootkit sets the target pin to low (False) without the Codesys runtime environment being aware of this change. The rootkit also causes the Codesys runtime environment to read the rootkit's desired value from pin 24 instead of the real value.

We implemented our rootkit as an LKM. LKMs allow us to access the kernel space. Once the module is loaded, it checks the CPU information of the machine and match it against a hard-coded list of CPUs and their I/O memory ranges. As mentioned above, the I/O addresses of the BCM2835 processor begin at 0x20200000. Using this information, the rootkit looks in the OS page table for a process PID and a virtual address that are mapped to the physical address 0x20200000. Because our rootkit already knows which pin to target (in our case, pin 22), it calculates the correct register address for the SET, READ, and Input/Output Mode registers. To control the write operations, the rootkit inserts the virtual address from the SET register associated with pin 22 (0xB6FCD034) into the BCM2835 processor's debug register and install its custom interrupt handler. Immediately after the Codesys runtime environment requests a write operation to pin 22 (for example, let us assume that it wants to write a value of 0 into the SET register to turn off the LED), our custom exception handler is called. This exception handler does not alter the content of the write request sent by the Codesys runtime environment. The exception handler instead executes one instruction and changes the state of pin 22 from output mode to input mode by writing the values of "000" to bits 6 to 8 of the Input/Output Mode register in 0xB6FCD01A. Immediately after changing the state of pin 22, the processor allows the Codesys runtime software to execute its command (which is to write a value of 0 to the SET register for pin 22). Codesys successfully writes this value to the register and returns a result of success to the Codesys runtime software. However, the processor does not apply the write operation to the pin as requested because the pin's state has been switched to input mode. At this point, our rootkit has full control over the Codesys I/O operations and can freely decide whether to allow an I/O state change.

The process of manipulating a read operation on pin 24 is almost the same. The rootkit inserts the virtual address from the READ register associated with pin 24 into the debug register and install its custom exception handler. Once Codesys accesses the virtual address from the READ register to read the value from the I/O pin, the exception handler executes. However, in this case, the exception handler executes two instructions instead of one, as in the write operation manipulation. The exception handler first sets pin 24 as an output pin by writing the values "100" to the related bits of the Input/Output Mode register and then writes the desired value for pin 24 into the SET register. The rootkit then returns the execution to the control of the Codesys runtime environment. The Codesys runtime software then reads the rootkit's desired value instead of the real value.

Our rootkit was able to successfully alter the physical process described above. We modified our physical process by allowing the Codesys runtime to turn on and off the LED on pin 22 only every ten seconds instead of every five. Additionally, the read manipulation performed by the rootkit rendered the button in our physical layout ineffective. We could hold the last state of the LED by giving fake read input values to the runtime and make the runtime assume that someone pressed the electrical button while it was not the case.

VII. Discussions

As discussed in Section V, the I/O attack cannot be detected by Autoscopy Jr. or Doppelganger. We verified this claim by testing the I/O attack against the current implementation of Autoscopy Jr. Unfortunately, the authors of Doppelganger were unable to provide us with their implementation for our test. Autoscopy Jr. does not detect this attack because no data hooking is performed in an I/O attack. We argue that Doppelganger is also unable to detect I/O attack because no modification of the static parts of the memory (e.g., code hooking) is performed in such an attack; instead, as described in Section VI, the entirety of the malicious code is loaded into the dynamic memory, which is not monitored by Doppelganger.

There are several characteristics of I/O attack that should be considered regarding their practical implementation. Therefore, in this section, we first discuss the performance overhead incurred by an I/O attack and then consider the hardware knowledge required for such an attack. Finally, we discuss about detection techniques that may be effective at detecting attacks leveraging dynamic memory and avoiding traditional function hooking, such as our I/O attack.

A. Performance

Embedded devices typically have limited resources for the operations they execute. This is the case for PLCs as well. While in general, embedded devices performance overhead is not an issue for the attacker, it can be when a PLC controls processes that are timely critical. If in such processes the performance overhead causes significant delay in the I/O speed, it can uncover the attack. We evaluated the performance of I/O attack on our selected hardware (Raspberry Pi model 1 B). Regarding CPU overhead, based on our evaluation, an I/O attack on average incurs 5% CPU overhead for the manipulation of write operations and 23% CPU overhead for the manipulation of read operations. Read operation manipulation imposes a higher CPU load for two reasons. First, the PLC runtime environment reads the values from the I/O multiple times per second, thereby significantly increasing the CPU overhead, whereas for write operations, the number of I/O write operations depends only on the logic (in our case, every five seconds). Second, read manipulation requires two instructions (setting the pin to output mode and writing to it), whereas write manipulation requires only one instruction (setting the pin to input mode). Figure 7 depicts the CPU overhead incurred by the manipulation of read and write operations in an I/O attack. The additional CPU overhead is not an important concern for the attacker, but it creates anomalies in the power consumption of the victimized device.

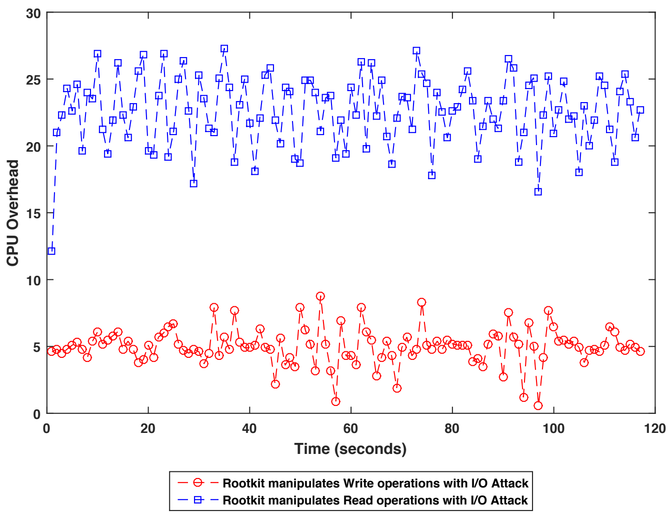

Fig. 7. CPU overhead in an I/O attack

To understand the impact of an I/O attack on control operations, we evaluated the I/O speed fluctuations in our selected setup (Raspberry Pi with Codesys runtime running our sample logic). Figure 8 depicts the fluctuation of the I/O speed with and without our rootkit implementation. On average the speed where our hardware could write to the I/O (without our rootkit) was 3.97 milliseconds. When the rootkit manipulates the I/O (intercept the I/O write operation and write the same value), the average speed of the I/O increased to 4.01 milliseconds.

The difference in I/O speed with and without rootkit is insignificant. Additionally, in a normal state (no rootkit operating), the I/O speed has a similar fluctuation to when our rootkit is executing an I/O attack.

B. Hardware Knowledge

In Section VI, we used the physical addresses of the I/O pins to find their mapped virtual addresses. Moreover, in our

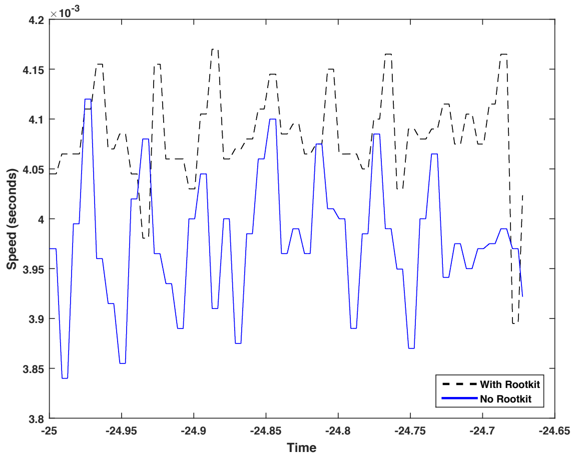

Fig. 8. I/O speed with and without rootkit

rootkit implementation, we had knowledge of all physical I/O register addresses. However, this is not the case for all types of processors. For example, certain PLC processors are proprietary. In this case, an attacker needs to perform the additional step of determining the physical addresses of the I/O pins of his interest. However, this necessity does not stop state-sponsored attackers. Detecting the I/O addresses that are used in either drivers or applications is straightforward. Unix-based operating systems provide I/O address ranges in /proc/modules for kernel drivers or in /proc/$pid/maps (where $pid is the PLC runtime process ID) for applications for I/O mapping. Nevertheless, detecting the I/O register addresses is a complicated task. Again, attackers who wish to target PLCs to attack critical infrastructures will investigate their targets sufficiently to determine this information. One solution for obtaining this I/O register information is to first decompile the available PLC logic within the PLC memory and search for I/O read/write operations and then monitor the read/write operations involving the mapped addresses retrieved from the OS (e.g., /proc/modules or /proc/$pid/maps). An attacker can begin looking for the I/O input/output mode registers by monitoring the PLC runtime environment when it is starting up. Additionally, from the decompiled logic, the attacker can be aware of the timing of the cycle of read and write operations in a specified I/O memory range. By monitoring read/write operations in that memory area (e.g., using debug registers), the attacker can identify the I/O read/write registers.

C. *Possibility for Race Condition*

There is a small chance that a race condition happens during the read manipulation of the I/O. For example, assume that we have a sensor connected to an input enabled pin in the PLC. If this sensor updates the value of the pin right after the rootkit does, then the PLC runtime reads the actual value of the I/O instead of the attackers intended value. This race condition can lead to a failure in the read manipulation operation of the I/O attack.

D. *Direction for Possible Solutions*

The I/O attack we designed reveals the limitations of current host-based solutions and the necessary characteristics of a possible future detection technique. In this section, we describe two research directions for possible detection techniques in embedded devices such as PLCs that can discover attacks based on dynamic memory. These two detection mechanisms form the basis of our future research concerning host-based detection techniques for embedded devices.

a) Dynamic memory verification: The first possibility for a solution consists of monitoring the behavior of the dynamic memory (heap) and memory areas with dynamic contents. In our I/O attack, we used TLS, which is a type of dynamic content memory. OS allocates the TLS memory region with a fixed size during kernel or application startup. However, unlike its static size allocation, part of the content of the TLS changes dynamically. This part of the TLS can include the I/O values that OS mapped into the TLS and we classified it as dynamic content memory. Obviously, monitoring dynamic memory or dynamic content memory by creating checksums of it is not practical since it is rapidly changing. Therefore, we propose to create a set of heuristics to monitor such memory. For example, these heuristics can be numbers of read and write operations into the dynamic memory in a period. By comparing the runtime information with these sample heuristics (or other similar heuristics), we might be able to detect possible dynamic memory attacks (e.g., heap overflows or I/O attack).

b) Control-Flow Integrity: We propose our second solution based on CFI. Autoscopy Jr. detects function pointer hijacking within the kernel. However, Autoscopy Jr. allows functions that are not listed in TLL entries to be executed. An attacker can therefore create a new malicious function that can have similar functionalities of functions within TLL, but in the attacker intended way, and call it. Since Autoscopy Jr did not monitor such malicious function during the learning phase (because it never existed during the learning phase), Autoscopy Jr. ignores it. For example, in an I/O attack, the functions that manipulate the I/O Input/Output Mode register can be called without facing any challenge from Autoscopy Jr., since it was never exist prior to attack.

To overcome this problem (and similar ones), we recommend the creation of a system for backward-edge CFI monitoring for PLCs. We use backward-edge CFI since we want to use the design features of embedded processors such as ARM to our advantage and reduce CPU overhead. ARM processor has a specific register for return addresses of the functions that is called Link Register (LR). With LR, instead of monitoring every instruction for direct and indirect jump or calls, we only monitor the LR register. Therefore, we suggest creating a Control Flow Graph (CFG) of the PLC runtime and recording all possible return addresses of the runtime. During the runtime execution, we can compare this graph with the values of the LR register to detect possible control-flow attacks. The system can raise an alert once there is a violation in the expected control-flow. For further performance improvement in our CFI approach and reduce the chances of attackers to execute a ROP attack we can find available ROP gadgets in the system and use them in our CFI approach. In ARM platform, this can be a practical approach since attacker has significantly less available gadgets compared to Intel architecture due to memory alignment differences.

VIII. Conclusions and Future Work

In this paper, we first looked into the current state of host-based detection techniques for embedded devices, with a particular focus on programmable logic controllers. We found that current practical host-based intrusion detection techniques for embedded devices suffer from three major shortcomings. First, they completely ignore the control of dynamic memory when verifying memory contents. Second, they do not apply effective practical control-flow measures due to performance limitations. Finally, they mostly rely on static references to protect embedded devices.

In the second part, we have proposed a new type of attack that leverages these weaknesses, and we have shown that it can be used by adversaries to manipulate the physical process in a way that the PLC runtime and the SCADA applications are unaware of the manipulation. This makes the attack interesting and relevant since current detection techniques are not effective against this new type of attack or any type of attack that exploits the weaknesses discussed in the Section IV.

We now plan to investigate in more detail the opportunities offered by the monitoring techniques we briefly sketched in Section VII. In particular, we will focus on the protection of dynamic memory and improving control-flow integrity in embedded devices. We believe that in any attack against embedded devices, control-flow integrity and dynamic memory verification measures will pose a notable hindrance to attackers, significantly reducing their success rate.

References

[1] F. Abad, J. van der Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, "On-chip control flow integrity check for real time embedded systems," in *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, Aug 2013, pp. 26–31.

[2] F. Adelstein, M. Stillerman, and D. Kozen, "Malicious code detection for open firmware," in *18th Annual Computer Security Applications Conference, 2002. Proceedings*, 2002, pp. 403–412.

[3] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, S. Merz and J. Pang, Eds. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516650

[4] Z. Basnight, J. Butts, J. L. Jr., and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76 – 84, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874548213000231

[5] D. Beresford, "Exploiting Siemens Simatic S7 PLCs," in *Black Hat USA*, 2011. [Online]. Available: http://www.cse.psu.edu/~sem284/cse598e-f11/papers/beresford.pdf

[6] D. Beresford and A. Abbasi, "Project IRUS: multifaceted approach to attacking and defending ICS," in *SCADA Security Scientific Symposium(S4)*, 2013. [Online]. Available: http://vimeopro.com/s42012/s4x13/video/58983658

[7] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[8] A. Cui, "Red ballon security." [Online]. Available: http://www.redballoonsecurity.com

[9] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *NDSS*, 2013. [Online]. Available: http://ids.cs.columbia.edu/sites/default/files/ndss-2013.pdf

[10] A. Cui and S. J. Stolfo, "Defending embedded systems with software symbiotes," in *Recent Advances in Intrusion Detectio: 14th International Symposiumn*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Springer, 2011, pp. 358–377.

[11] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 133:1–133:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2596656

[12] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium*, 2014. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-davi.pdf

[13] DigitalBond, "3S CoDeSys, Project Basecamp," 2012. [Online]. Available: http://www.digitalbond.com/tools/basecamp/3s-codesys/

[14] ——, "WAGO IPC 758/870, Project Basecamp," 2015. [Online]. Available: http://www.digitalbond.com/tools/basecamp/wago-ipc-758870/

[15] L. Duflot, Y.-A. Perez, and B. Morin, "What if you cant trust your network card?" in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 378–397.

[16] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, 2011.

[17] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, ser. SecuCode '09. New York, NY, USA: ACM, 2009, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1655077.1655083

[18] I. Fratrić, "ROPGuard: Runtime prevention of return-oriented programming attacks," 2012. [Online]. Available: http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf

[19] ICS-CERT, "Schneider electric modicon quantum vulnerabilities (update b)," 2014. [Online]. Available: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-12-020-03B

[20] V. M. Igure, S. A. Laughter, and R. D. Williams, "Security issues in SCADA networks," *Computers & Security*, vol. 25, no. 7, pp. 498–506, 2006.

[21] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, 2004.

[22] M. LeMay and C. Gunter, "Cumulative attestation kernels for embedded systems," *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, June 2012.

[23] Z. Liang, H. Yin, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in *Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008. [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/hookfinder_ndss08.pdf

[24] S. McLaughlin and P. McDaniel, "SABOT: Specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 439–449. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382244

[25] S. E. McLaughlin, "On dynamic malware payloads aimed at programmable logic controllers," in *HotSec*, 2011. [Online]. Available: https://www.usenix.org/legacy/event/hotsec11/tech/final_files/McLaughlin.pdf

[26] Microsoft Corporation, "Enhanced mitigation experience toolkit," 2014. [Online]. Available: https://www.microsoft.com/emet

[27] O. S. V. D. (OSVDB), "D-link dir-605l wireless n300 cloud router captcha data http request parsing remote buffer overflow," 2012. [Online]. Available: http://www.osvdb.org/86824

[28] V. Pappas, "kBouncer: Efficient and transparent ROP mitigation," 2012. [Online]. Available: http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf

[29] D. Peck and D. Peterson, “Leveraging ethernet card vulnerabilities in field devices,” in *SCADA Security Scientific Symposium*, 2009, pp. 1–19.

[30] D. G. Peterson, “Project basecamp at s4,” *Digital Bond Blog, Digital Bond, Sunrise, Florida*, 2012. [Online]. Available: www.digitalbond.com/2012/01/19/project-basecamp-at-s4

[31] pt, “Oops, I hacked my PBX. Why auditing proprietary protocols matters,” *28th Chaos Communication Congress*, 2011. [Online]. Available: https://events.ccc.de/congress/2011/Fahrplan/attachments/2023_oops_i_hacked_my_pbx.pdf

[32] Rapid7, “D-link hnap request remote buffer overflow,” 2014. [Online]. Available: http://www.rapid7.com/db/modules/exploit/linux/http/dlink_hnap_bof

[33] ——, “Linksys wrt120n tmunblock stack buffer overflow,” 2014. [Online]. Available: http://www.rapid7.com/db/modules/auxiliary/admin/http/linksys_tmunblock_admin_reset_bof

[34] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, “Intrusion detection for resource-constrained embedded control systems in the power grid,” *International Journal of Critical Infrastructure Protection*, vol. 5, no. 2, pp. 74–83, 2012.

[35] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, “Evaluating the effectiveness of current anti-ROP defenses,” in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer, 2014, pp. 88–108.

[36] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, “SWATT: SoftWare-based attestation for embedded devices,” in *2004 IEEE Symposium on Security and Privacy. Proceedings*, May 2004, pp. 272–282.

[37] P. Traynor, K. Butler, W. Enck, P. McDaniel, and K. Borders, “Malnets: Large-scale malicious networks via compromised wireless access points,” *Security and Communication Networks*, vol. 3, no. 2-3, pp. 102–113, 2010.

[38] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, “Dynamic hooks: hiding control flow changes within non-control data,” in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 2014, pp. 813–828.

[39] S. Wegner, “Security-analysis of a telephone-firmware with focus on backdoors,” Bachelor’s thesis, Ruhr-Universität Bochum, 2008. [Online]. Available: https://git.fabrik17.de/mrgitlab/embedded-multimedia/raw/437afd92da4b438f95fa3efad28564a9d0baffbd/Dokumentation/_thesis_template.pdf

[40] R. Wightman, “Project basecamp at s4,” *SCADA Security Scientific Symposium*, 2012. [Online]. Available: https://www.digitalbond.com/tools/basecamp/schneider-modicon-quantum/

[41] F. Zhang, H. Wang, K. Leach, and A. Stavrou, “A framework to secure peripherals at runtime,” in *Computer Security-ESORICS 2014*, M. Kutyłowski and J. Vaidya, Eds. Springer, 2014, pp. 219–238.